Deadlock

Release 0.1

Nov 14, 2016

Contents

Project Deadlock is a system that controls access to a number of points of access (e.g. doors, appliances) using RFID cards. Deadlock is designed for security and reliability, assuming untrusted and unreliable network. Unlike existing commercial solutions, Deadlock is fully open-source and open-hardware, and designed to be flexible, maintainable, scalable, and cost-effective. We provide tools and expose all interfaces and components, making Deadlock easy to integrate with existing systems and customize to the needs of the user.

CHAPTER 1

Table of contents

1.1 Introduction

1.1.1 Project Deadlock: RFID Physical Access Control System

... for opening doors and more.

This project aims to create an open-source system for our university that will allow ISO/IEC 14443a compatible cards (like International Student/Teacher Identification Cards) to be used for accessing doors. We want this system to be:

- **Reliable and secure.** The access will be granted when and only when it should be.
- **Easy to use and maintain.** Access rules configuration should be fast and simple. Potential failures of certain components should be quick and easy to fix.
- **Fast to deploy and configure.** Minimal overhead should be included in deploying this system (cables, etc...)
- **Available.** Hardware parts should be cheap to manufacture and components for them should either be available in the future or painlessly replaceable by their newer alternatives.
- **Extensible.** The whole system, including the hardware components, should be extensible to perform some function not needed right now in the future.
- **Done.** The project is about a year overdue. We *really* want this system to be done.

(I am sorry, I sound like a businessman. That's not intentional and I'll stop that right now.)

We will provide a complete system for managing access rules (integrated with the university's electronic info system), and the embedded hardware and software.

(OK, I mean it this time. No more businessman)

For a more thorough discussion of the requirements see [[Requirements]].

To better understand these you can read about the [[motivation and history]] of the project here.

1.1.2 Under development

The whole system is under active* development. Some things we are talking about here do not exist yet, do no longer exist, have never existed or may never exist.

(* for the proper definition of the word active. But hey, we are doing what we can!)

Here is a Trello board you can use to monitor the development progress: https://trello.com/b/eUShodyG

1.2 Requirements specification

Project Deadlock aims to create a complete system to allow cards compatible with the ISO/IEC 14443a standard (commonly known as *RFID cards*), such as International Student/Teacher Identification Cards, to be used to unlock doors and access other electronic equipment (hereafter *points of access* or *PoA*).

For this system to be useful at our university, Deadlock must meet the requirements outlined below.

1.2.1 Trustworthiness

As Deadlock may be used to protect valuable resources, such as computer rooms or labs, it must allow access when and only when it should. We must provide the user with reasons to trust this promise.

Reliability

Points of access should be accessible even when things go wrong; specifically partial power or network outages must not prevent the system from allowing access, nor cause loss of access logs. Temporary server failure must cause no problems. Furthermore, the design and implementation should allow for a simple failover mechanism.

Security

Deadlock must not allow illegitimate access. To protect privacy, logs or card IDs must not leak. We cannot assume a private communication channel. Therefore all communication in both directions must be authenticated and kept confidential.

1.2.2 Practicality

Deadlock must be an effective solution for our use case. This must hold even if the use case changes in the future.

Extensibility

In order to be prepared for the future, and to make incremental development possible, all software and all hardware must be modular, with well-defined interfaces, and extensible.

Functions not implemented in the first iteration, but expected to be added in the future, are

- arbitrary communication with the card (RFID cards are capable of complex actions, such as cryptographic verification of identity, or local data storage)
- controlling arbitrary appliances, not just door locks,
- WiFi module (for cases when power is available but Ethernet is not).

Ease of development

In the future Deadlock will likely be developed and maintained by students, not full-time developers. Therefore the codebase must be simple, easy to understand and change, the tools and libraries must be easy to use, and the overhead of introducing a new developer to the project must be minimal. When possible, general, well-known solutions should be used instead of solutions developed in-house.

Ease of use

Setting up access rules should be simple and convenient. This should not come at the expense of generality. Synchronization with the university's electronic information system is required, so that card information and groups like "CS teachers" or "PhD students" can be imported automatically.

The system should notify the operator if human intervention is required, but simple tasks and predictable issues should be handled automatically.

Ease of deployment and maintenance

Deployment should be simple and with minimal overhead. On the hardware side, it should be possible to leverage existing infrastructure in order not to need extra cables for communication or power. On the software side, importing data from existing sources (such as our university's Academic Information System) should be possible. Replacing any failed components should be quick and should not require substantial training. The system should check its state and automatically fix whatever can be fixed automatically, e.g. reboot a device if it gets locked up.

Availability

Hardware should be cheap to manufacture and components should either be available in the foreseeable future or painlessly replaceable by newer alternatives.

In order to make Deadlock as available as possible, we release both the hardware schematics and the software to the public under the MIT license.

1.2.3 Further considerations

Power outage behavior

In case of a power outage at entrance/exit PoAs, some doors should stay locked (to avoid the risk of breaching security), and some doors should open (e.g. emergency exits). Both can be supported by using different lock hardware and changing configuration.

Emergency open

The hardware locks on entrance/exit PoAs must support manual opening and locking by authorized personnel. This is useful in emergencies.

1.3 System overview

1.3.1 Key design principles

Modularity

Separating the functionality into independent modules with well-defined, simple, minimal interfaces simplifies development, makes the design much easier to grasp and minimizes the learning curve for a new developer. Things are better if one knows where to look for certain functionality (and where not to).

Principle of Least Astonishment

"People are part of the system. The design should match the user's experience, expectations, and mental models." [@PrinciplesDesign] If the design, implementation or behavior of a part of the system is obscure enough to surprise you, it should be redesigned.

We should prioritize predictability of the parts of the system which are hard to observe (such as the embedded devices), so that they do not unpleasantly surprise the user.

State is ugly

State adds complexity to a system: if something has state, it has code managing it, and the developer must keep track of how the internal state influences the system. Also, if the system is stateful, it complicates failure recovery, as the state must be replicated or otherwise re-creatable. Therefore when at all possible, the components of Deadlock should have little internal state and depend only on explicit, well-defined, easily replicated data.

1.3.2 Main components

The system consists of a central server and a number of controllers. Each controller serves a single point of access, holds a copy of the access rules and evaluates them locally. The server provides controllers with rules updates and collects access logs.

Server

The server is the "manager" of the whole system. It holds the authoritative version of the access rules, collects access logs and provides software updates and time synchronization for the other devices. It monitors system state (and reports it to the management UIs).

Except for the contents of the database, it is entirely stateless. This simplifies the code and makes replication and failover trivial.

For details see the server documentation.

Controller

The controller controls its associated point of access (e.g. unlocks its door). It takes actions (such as opening the door or logging) based on events observed (such as a card being presented or the door being opened). It periodically contacts the server, reporting its status and checking for updates.

The controller is "almost stateless" – logs are sent to the server, and rules and firmware updates can be retrieved from the server. Therefore a device can be swapped simply by writing the correct device ID and encryption key to either the device or the database.

We have developed an extensible embedded device that can function as a controller. It is tailored to the "door lock" use case.

For details see the controller documentation.

Reader

The user-visible box at PoAs that reads RFID cards and provides visual and auditory feedback about whether access is granted. Up to two card readers may be attached to one controller. A library to interface with our readers is provided, so they can be used independently of our controller.

For details see the reader documentation.

Hardware

The server is hardware-agnostic – it runs on anything with networking and a Python environment. Deployments are expected to use generic server hardware.

We have designed and built custom hardware for the controllers and readers. We focused on making it available and future-proof, extensible, and cheap. The schematics and other documents are available on Github.

In order to simplify installation, we have attempted to leverage existing infrastructure wherever possible: we use Ethernet for server/controller communication, adding optional Power over Ethernet, so we don't require any extra cables. Optionally, we can add a WiFi module to the controller for cases where electricity is available but connectivity is not. We even designed our reader boxes and connection cables to be easy to customize, so that they can be made compatible with existing holes in walls. (For example, instead of using an expensive cast for the reader boxes, we make them from several layers of Plexiglas that can be cut individually. The source files for the cut pattern are available and easy to modify.)

1.3.3 Access rules

The decision whether to grant access is a function of user identity, point of access, date, time, and day of week. The access rules are designed to be simple without sacrificing generality. For details see the access rules documentation.

1.4 Access rules

1.4.1 Generality vs. convenience

The key observation when designing the access rules is:

Generality comes at the cost of complexity. For any given application, most rules will look the same, and therefore if the rules are general, then they will be unnecessarily complex in the typical use case. Most of the time, the user will be annoyed by inputting similar rules every time, instead of making use of the generality.

Because of this problem, we have decided to create two distinct layers of access rules: a low-level layer, which is general and simple, and a domain-specific high-level layer, which is optimized for the typical usage in the given domain. The high-level layer builds on the primitives provided by the low-level layer, and different high-level rules should be developed for different use cases (such as campuses vs. hotels). This allows for flexibility and convenience at the same time.

In order to support both the typical use case and the unique snowflakes in a single installation, high-level rules implementations are expected not to assume anything about the rules installed in the system – they are not allowed to carelessly delete existing rules, or assume only rules they know about exist. They should display the low-level representation for rules they cannot interpret in their high-level model.

To facilitate this cooperation between high-level and low-level rules, and to ensure consistency, we have come up with the notion of a *ruleset*: every rule in the system is tagged as belonging to exactly one ruleset, and the high-level layer can create, update or delete only whole rulesets, not individual rules. Operations on rulesets are atomic. An application implementing the high-level rules should operate only on rulesets created by that application. A mechanism for enforcing this restriction exists. (This is implemented in our DBMS of choice, PostgreSQL, by the row-level security mechanism).

The low-level, internal rules must be generic enough to support any use case, yet easy to compile by both computers and people.

1.4.2 Internal rules format

In order to cover all possible use cases, the straightforward approach is to allow access rules to be specified as any Boolean formula over identities, access points and time specifications. However, this brings the following problems:

- it is hard for humans to quickly reason about the result of any given query
- complete evaluation on every query is necessary, which might be costly in memory or time; it is impractical to pre-compute much for large inputs
- for offline functionality, the evaluation logic and all data required for evaluation need to be embedded in the controllers, which violates the "keep embedded devices simple" design principle;
- a small change in input data or formulas can have arbitrarily large effects, which hinders attempts at both understanding why something happens and pre-computation.

In order to avoid these problems, we have instead chosen the following model:

Every access point is of exactly one *type*; for each type, rules that match a *time specification* and an *identity expression* to an *Allow* or *Deny* response may be added. Rules are strictly ordered by priority.

The evaluation flow, as depicted below, is as follows:

1. Find this AP's type, select its rules.
2. Select rules with matching time specification.
3. Select rules where this identity matches the rule's identity expression.
4. Select the (single) rule with the highest priority.

This selects a single rule, which unambiguously allows or denies access.

TODO picture

Identity expressions

An identity expression is a (restricted) Boolean formula over identities only, and it implements a generalization of access control by groups.

Implementing general Boolean formulas (e.g. using AND, OR and NOT operators) would be possible, but to support NOT we would have to either store the complement, which may require a lot of memory for a small input, or make the computation less straightforward, which clashes with keeping controllers simple. Therefore identity expressions use the INCLUDE and EXCLUDE operators, which are equivalent to set union and set difference. These are equivalent (even in expression complexity) to general Boolean formulas as long as the set of "interesting" identities is given (which it is, as "any ID whatsoever except for this one" is not a useful rule).

Therefore, we define identity expressions as$\textit{expr} \defeq \text{INCLUDE }x_1, \ldots, \text{INCLUDE }x_m,, \text{EXCLUDE }x_{m+1}, \ldots, \text{EXCLUDE }x_n$where $x \defeq \textit{expr}, |,\textit{identity}$with the semantics "union of all INCLUDEd sub-expressions minus union of all EXCLUDEd sub-expressions".

Rationale for the separation

Splitting rule evaluation into identity expressions and time+place expressions means that rules are easier to evaluate: a human (or a computer) can evaluate the two independently, and "why does this happen" questions are easier to answer. Moreover, in this way classes of equivalence on inputs are easier to find, as in this model a single time+place rule matches a single identity expression instead of arbitrary combinations. This makes it practical to pre-compute some rules, and implement re-computing this incrementally on change.

1.4.3 Quick access querying: "in expression" relation

Typically, rules will be queried often (especially when creating local rules databases for controllers) and changed infrequently. Therefore we can save work and time by pre-computing some information. Currently, we assume that in a typical deployment there will be few rules and multi-level identity expressions. Therefore we pre-compute an "in identity expression" relation – for every identity (i.e. for all leaves of the expressions) we climb the expression tree (or, in fact, DAG) and save the $(\textit{identity}, \textit{expression})$ tuple when the identity is included by an expression (taking into account the INCLUDE/EXCLUDE operations). As the expressions are acyclic, whenever we need to INCLUDE/EXCLUDE a sub-expression, we can re-compute expressions in the order of dependencies (and therefore exactly once).

In order to select the rule applicable for a given access query according to section \ref{rulefmt}, in step 3 we simply select rules where an $(\textit{identity}, \textit{expression})$ tuple exists. Similarly, when creating the local database for controllers, only the flattened relation, not the original hierarchy, is used.

When an identity expression changes, it is easy to incrementally re-compute only the affected parts: we simply search the DAG, re-computing nodes as we visit them.

See section \ref{impl:inexpr} for notes on the implementation of re-computation.

1.4.4 Local evaluation on embedded devices

The local database copy on the controllers builds on this two-level approach of separating identity expressions and time specifications. Note that any controller serves a single point of access, and therefore the "where" part of the rules is already taken care of – every point of access knows only about rules belonging to its type.

The server listens for "rules changed" notifications from the database and rebuilds the controller-specific local databases when needed. The specific format of the local databases is out of scope of this thesis.

1.4.5 Integration with existing systems

As required by section \ref{requirements:ease-use}, data may be imported from other systems, and transparently "patched" into access rules. This is done via an application that generates flat identity expressions of the form $X \defeq [\text{INCLUDE}; \textit{person}_1, \text{INCLUDE}; \textit{person}_2, \ldots]$ for every group X that needs to be imported. These groups are marked and considered to be primitives, and they may be modified only by creating a group Y that INCLUDEs X and further INCLUDEs or EXCLUDEs what needs to be adjusted in the imports. In this way when the underlying data changes, the "patches" will not be disturbed.

1.5 Developer Intro

1.5.1 Per-subproject considerations

Make sure to read all READMEs in the repository that you are interested in.

1.5.2 Repositories

The project has many components, and that means we have many repositories for them.

Server and general

- `server`. This repository hosts the source code for the server. It is also main repository for the documentation of the whole system. Design issues and issues not specific against any single component should be opened in this repository.

Other server-related repositories do not exist yet.

Controller

- `controller-hw`. This repo will contain schematics and PCB designs for the Controller.
- `controller-sw`. Software for the Controller.
- `temp-controller.py`. We needed to deploy 2 prototypes right away, so this is a temporary jerry-rigged controller software to be used on Raspberry Pi. Will be obsolete soon.
- `libmfrc522.py`. This is a library for controlling the MFRC522 module used in the Reader. It is the module that actually reads cards. It is used in temp-controller, because the deployed prototype is using an older concept of the reader, which cannot read cards on its own, it just mediates communication between the MFRC522 module and the Controller. In the new concept it will not be needed, and will be obsolete soon.

Reader

- `reader-hw`. Schematics and PCB design of the Reader.
- `reader-sw`. Software for the Reader.
- `libreader.py`. This is a Python library for interfacing with the Reader. It was made for the temporary controller, bud will be kept up-to-date as it allows almost any Python-capable system to use the Reader. It also allows for easier debugging and development of the Reader.

Management interfaces

- `webui`. The web interface to Deadlock – a console for managing controllers, points of access, and access rules.

Utilities

- `hw-testing`. This repo contains library for testing STM32 MCU powered embedded devices, like Reader and Controller. It also contains tests for these boards. New tests are written easily, and it is used in development of the system, as well as during the manufacturing for final checkout of the hardware.

1.6 Rationale

Design decisions are explained here. If you intend to hack on Deadlock, or re-use some of it's components (which you are welcome to do) it may help you to read these.

This is also the place we user for rubber-duck debugging.

Contents:

1.6.1 Design of modular protocol stack for Desfire cards

What is a Desfire card?

Let's think of it as of a contactless memory card with access control.

According to this, DESFire card is a `ISO/IEC 14443A` compatible card which uses optional `ISO/IEC 7816-4` commands. It can store several Applications with different access permissions, with several Files in each Application.

What protocols are in play?

So, Reader must be equipped with `ISO/IEC 14443A` (part 1 and 2) compliant Reader (reffered to as PCD in `ISO/IEC 14443`). In our case, this is a `MFRC522` module connected over SPI. Driver for this module must be implemented in software.

`ISO 14443A` parts 3 and 4 must be implemented in the software in order to perform anticollision loop, select a card and activate it. According to the standard, a card is activated by performing an 'Anticollision' loop. This allows selection of a particular card (called PICC) from several cards which may be in the RF field generated by the PCD.

DESFire then has it's own set of commands. These can be sent either wrapped in `ISO/IEC 14443-4` data blocks or wrapped in `ISO/IEC 7816-4` APDUs. In addition, DESFire card supports several `ISO/IEC 7816-4` commands.

What do we need to do with the card?

The first iteration needs only to read UID of the card. Therefore everything above `ISO/IEC 14443-3` is unnecessary. However, it is almost certain that in the future we will want to do something more, so the system should be designed to allow easy addition of `ISO/IEC 14443-4`, `ISO/IEC 7816-4` and DESFire protocol library (on top / using encapsulation defined by either of these libraries).

First iteration is using `MFRC522` module connected over SPI. It is quite probable, however, that in the future the module may be changed for other type, or the same type can be kept but may be connected over some other interface (`MFRC522` supports SPI, I2C and UART).

This implies nice modular design. If done properly, parts of this stack may be reused in some other future projects.

Existing designs

The obvious thing to consider is to use some existing opensource implementation of this stack. Unfortunately most embedded libraries strictly adhere to design principle "fuck modularity, everyone will be using just one MFRC522 over SPI with Arduino reading only UIDs" and implement all the afromentioned funcionality in one file of spaghetti code. Unfortunately such implementations are unusable for us.

One very good implementation is `librfid`. Unfortunately this library is hard to integrate with ChibiOS and is not designed for embedded environment.

Card stack design

I've tried to take the most promising-looking existing implementation and modularize it, but found out that it would be easier to write it from scratch.

It was previously described that we need a modular design since we may be changing components. However, if we overdo it with modularity we will end up with hard-to-use hard-to-maintain code (e.g. everything-independent `MFRC522` library then requires design of platform-specific SPI / USART / I2C bindings and everything). A sane compromise must be found.

We are using and will continue using ChibiOS. The whole Reader application will use ChibiOS's facilities, will be multithreaded and this card stack should be designed with this in mind. It will have to integrate nicely with existing HAL of ChibiOS (which is by itself really nicely porable and can be used independently of ChibiOS/RTOS, since OSAL (Operating System Abstraction Layer) is placed between the RTOS and HAL).

The stack shlould look like this (top to bottom):

- DESFire Protocol Library
- (optional) `ISO/IEC 7816-4` Protocol Library
- `ISO/IEC 14443-3` and `ISO/IEC 14443-4` Protocol Libraries
- `MFRC522` driver

The stack must support swapping of implementation of various layers and using several different implementation of layers simultaneously (for example, the device could have 2 different RFID modules connected, this library must allow these two to be used simultaneously). Nice modular separation will also allow for mocking various modules for testing purposes.

This modularity will be achieved by object-oriented design. Each lower layer will create a structure with pointers to functions of the given lower layer for the upper layer to use. This structure will have fixed common API and a single `void` pointer to a lower-layer specific structure containing data the lower layer needs. Upper layer will then call these functions from the object with the object itself as a first parameter. This will allow encapsulation and usage of several objects from different layers simultaneously. The upper layer may, if necessary, wrap this object to a object of it's own, to add custom data for working with this object and for exporting this object to even upper layers.

DESFire Protocol Library and `ISO/IEC 7816-4` Protocol Library

These are not needed at this point. However it is guaranteed that they can be written easily using `Card` objects provided by the `ISO/IEC 14443` Protocol Library.

`ISO/IEC 14443` Protocol Library

This library implements `ISO/IEC 14443` part 3 (Initialization and Anticollision) and part 4 (Transmission Protocol). It operates on top of an abstract `ISO/IEC 14443` PCD API, which the PCD driver implements (described bellow).

`ISO/IEC 14443` defines two types of cards: A and B. Only part A will be implemented, because we don't need the part B right now (we have no hardware capable of communicating with B-type cards). The library will however be written with part B of the standard in mind so that it can be easilly added later if needed. Names of functions related to A part of the standard will end with `A`, B-function names will end with `B` respectively, and common function names will end with `AB`.

Although the library itself tries to be stateless, Reader is a state machine and Card is a state machine, which respons differently to the same stimuli when in different state. State of these entities is therefore encapsulated in structures `Pcd` and `Card`. All functions contain state checks on these objects to prevent undesired behaviour. During initialization both `Pcd` and `Card` structures must be passed as parameters to functions, after the card is activated it is bound to a `Pcd` (until deactivation) and `Card` structure contains pointer to the `Pcd` structure, so only `Card` structure has to be used.

`ISO/IEC 14443A` defines a way to detect and communicate with multiple cards in the same RF field. All `ISO/IEC 14443A` cards must support mechanism for detecting multiple cards, however, mechanism for activating and communicating with several cards simultaneously is optional. If PCD wants to activate several cards at once the workflow is as follows:

- Anticollision loop runs, and when it finishes the PCD knows `UIDs` of all PICCs in the RF field.

- PCD can then request activation of some PICC based on it's UID, and it can assign it a CID. CID is a unique identifier (between 1-15 inclusive) of a PICC in the RF field.
- PICC then tells the PCD whether it supports the CID feature.
  - If PICC does support CID the PCD sends blocks containing this CID to this PICC and will not use this CID with other PICC while the current PICC is active. It also can't activate any PICC which does not support CID.
  - If PICC does *not* support CID the PCD will send blocks without CID and can't activate any other PICC while the current PICC is active.

The library will therefore obey these rules and:

- If there is no active card it will allow activation of any card.
- If there is an active card which doesn't support CID it will not allow activation of any other card.
- If there is an active card which supports CID it will allow activation of other card which supports CID (but no more than 15 cards in total).
- If there is an active card which supports CID it will not allow activation of other card which does not support CID.

The library will handle assigning CIDs internally. It will keep state in the Pcd structure.

When the card is selected optionally communication speed parameters may be negotiated. Request for negotiation, however, must be the first thing a PICC receives (CID is taken into account at this stage), because if it receives any other valid data block it will automatically disable the negotiation feature. Support of this feature is also optional. This library contains select functions which will select a PICC and assign a CID (if applicable). After this function is used on a card either set_parameters, set_best_parameters or set_default_parameters functions may be used. These will set the communication parameters for the card and remember them in the Card structure (so that everytime we communicate with a PICC the PCD can be reconfigured accordingly). After these functions are used normal communication with the PICC may start.

ISO/IEC 14443-4 defines a half-duplex request-response communication protocol. When a PCD sends a request it must wait either for the response or for the timeout to occur. It can't transmit anything to the card during this time. Although not forbidden by the standard, this library will also disallow sending two requests to two different cards (with different CIDs) at the same time, as when this happens there is a risk that the response of the first card will be lost. Not to mention the possible necessity of switching communication speeds for different CIDs. This library therefore exports function send_receiveAB. Use of this function is strongly encouraged as it simplifies the development. This function will block until either the response is received or an error (such as a timeout) occurs. Internally, this function uses semaphores for waiting, and therefore allows other threads to run (it does not busy-wait if underlying OSAL supports it).

For flexibility this library also provides functions send and receive. These functions return immediatelly and have appropriate locking in place so that send cannot be used when some other communication is in progress or receive has not yet been called. Using these complicates application code, so their use is discouraged.

API

- get_cards_in_fieldA(Pcd,Card*): This function performs an anticollision loop as defined in ISO/IEC 14443A and returns an array of Cards. The Card structure will contain an UID of the card. By design, after successfully performing an anticollision on a card the card transitions to an ACTIVE state, and that is a side-effect we don't really expect of this function, therefore after activating new cards it also HALTs them. This function also allocates new Card objects. If you don't use them, don't forget to free_cardAB them.
- construct_cardA(uid,uid_length): This function constructs a new Card structure.

- `select_cardA(Pcd,Card*)`: This function attempts to select a card. The Card structure may come either from `get_cards_in_fieldA` or may be constructed by user (useful if expected card UID is known in advance). Only `uid` and `uid_length` fields of the `Card` structure are taken into account, this function will properly initialize all other fields. This function will also obey all rules related to `CID` described above.
- `detect_and_select_single_cardA(Pcd,Card*)`: This function performs an anticollision loop as defined in `ISO/IEC 14443A` and activates card with the highest `UID` it finds. This function will not provide any further info about other cards in the field, however, it is a bit faster than using `get_cards_in_fieldA` and `activate_cardA`. This function is useful if it is reasonable to assume that only one card will enter the RF field at a time.
- `deactivate_cardAB(Pcd,Card*)`: This function will deselect previously selected card, freeing resources associated with this card and putting the card to a `HALT` state.
- `set_parametersA(Card*,transmit_speed,receive_speed)`: This function will configure the PICC to use these communication parameters. It will fail if either PICC or PCD doesn't support these parameters, or if the PICC doesn't support setting communication parameters in general.
- `set_best_parametersA(Card*)`: This function will set the fastest communication parameters supported by both the PICC and the PCD. If PICC does not support setting parameters in general this function succeeds and leaves the parameters at their default values (because, semantically, that is the best the card can do).
- `set_default_parametersA(Card*)`: This function will skip the parameters negotiation entirely and will just use the defaults.
- `send_receiveAB(Card*,buffer*,length)`: This function will send data to the PICC, wait for the response and return the response in the `buffer`. If the `buffer` is not big enough to hold the whole response only part of the response which fits the buffer will be returned and the rest can be obtained using `get_responseAB(Card*,buffer*,length)`. This function will block until the whole response is received.
- `sendAB(Card*,buffer*,length)`: This function sends data to the PICC. It will return immediatelly.
- `receiveAB(Card*,buffer*,length)`: This function will return immediatelly either the received response or error that the response has not yet been received.
- `receive_waitAB(Card*,buffer*,length)`: This function will return the received response and will block until the response is received.
- `free_cardAB(Card*)`: This functions `frees` the `Card` structure and all resources associated with it. Make sure it is not used afterwards.

Minor state-inquiring and plumbing functions are ommited for clarity.

Exports to other layers

This library provides standardised object `Card` for use with other layers. The `Card` object is a structure which contains pointers to functions `send_receiveAB`, `sendAB`, `receiveAB` and `receive_waitAB` and one `void` pointer which other layers shouldn't touch.

The purpose of the last `void` pointer is to point at the internal structure used by this library to keep state of the card and other info about the card which other layers should not have to know about. This allows for some other library to provide its own `Card` structures with completely different internal representation of state and everything.

It goes without saying that functions of this library are designed to work only on `Card` structures created by this library. If you pass `Card` structure created by other library as a parameter you will get what you deserve: undefined behaviour.

Memory management

This library wont `malloc` nor `free` any `Pcd` structures.

This library **caries sole responsibility** for `malloc`ating and `free`ing its `Card` structures. User **shall not** `malloc` his own `Card`s nor `free` any `Card`s allocated by this library. For each `Card` allocated by this library also internal structure for internal data is allocated. If the user would `free` the `Card` reference to this internal structure would be lost, resulting in a memory leak. Functions `construct_cardA(uid,uid_length)` and `free_cardAB(Card*)` should be used instead.

However, it is responsibility of the **user** to call `free_cardAB(Card*)` on `Card` structures he no longer wishes to use.

It is also responsibility of the **user** to `malloc` and `free` data buffers passed to various send and receive functions. This library can be compiled with `SAFE_BUFFERS` option, which will cause that whenever a function for sending data is called the data buffer is copied to the internal buffer before the function returns. If this library is compiled without the `SAFE_BUFFERS` option the passed buffer is used directly. This can save memory, but then **the user has to ensure that the transmit buffer is not modified nor freed until the response is received**.

Threading and thread safety

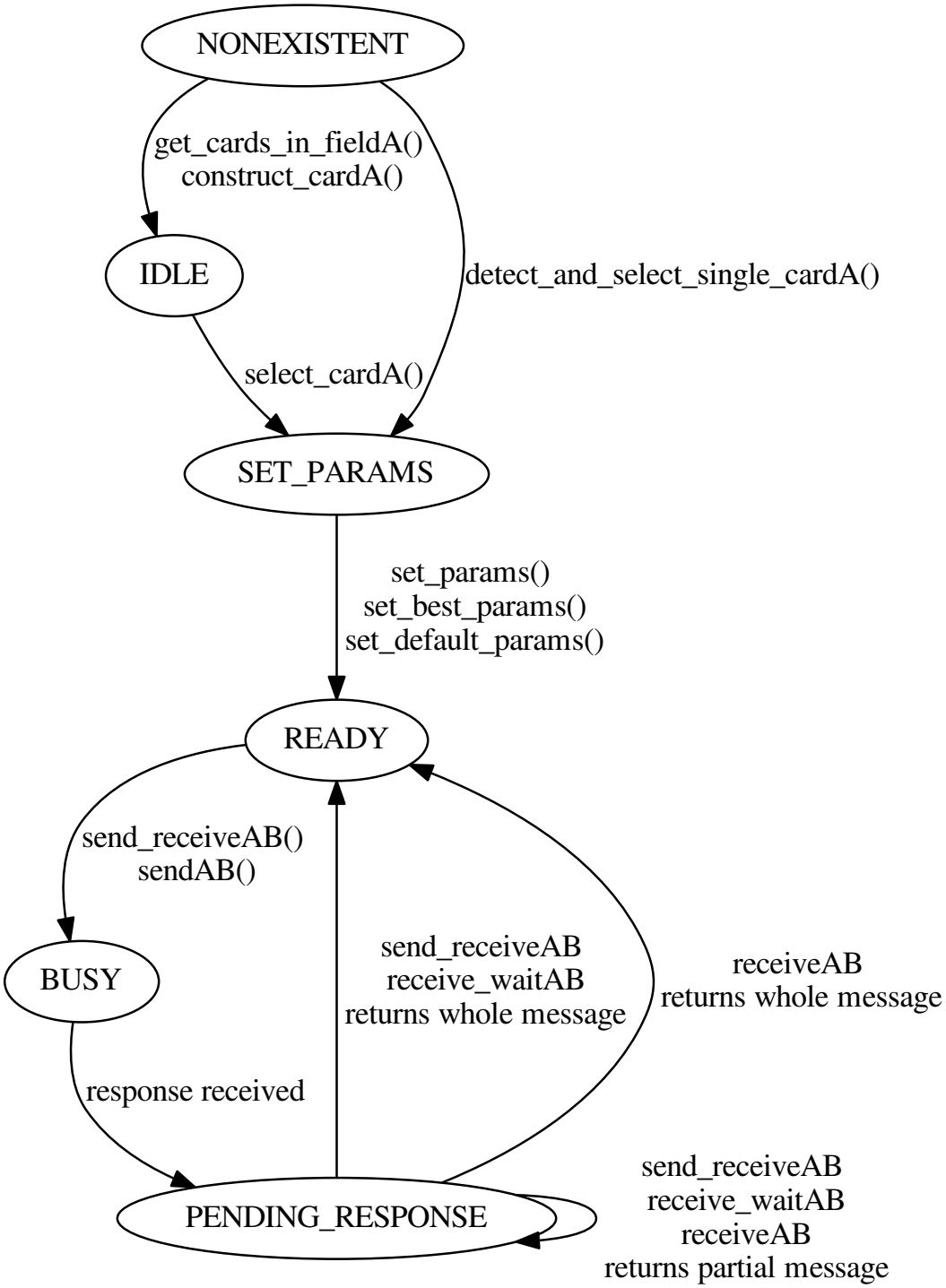
This library is designed to be used in a multithreaded enviroment. It won't create any threads for itself, however all functions are thread-safe (if underlying OSAL correctly implements locking primitives), and those that semantically cannot be used concurrently will return appropriate errors.

States and transitions

States and transitions of a `Card`

Function can be used only if it is indicated on transition edge here, otherwise the function will fail.

Functions `deactivate_cardAB` and `free_cardAB` were ommited for clarity. These can be used in any state (except `BUSY`) and will set the state to `IDLE` or `NONEXISTENT`, respectively.


NONEXISTENT
get_cards_in_fieldA()
construct_cardA()
IDLE
detect_and_select_single_cardA()
select_cardA()
SET_PARAMS
set_params()
set_best_params()
set_default_params()
READY
send_receiveAB()
sendAB()
BUSY
send_receiveAB
receive_waitAB
returns whole message
receiveAB
returns whole message
response received
send_receiveAB
receive_waitAB
receiveAB
returns partial message
PENDING_RESPONSE

Tests

ISO/IEC 14443 part 4 defines several example communication scenarios which could be used when designing compliance tests. However, implementing those tests is still an open problem.

Although in tests the Pcd can be (by the virtue of modular design) easily mocked the same cannot be said about the OSAL (abstraction on top of the ChibiOS/RT) which this library relies upon. Research will be done in this area.

MFRC522 driver

This layer presents upper layers with an abstract ISO/IEC 14443 PCD API and implements drivers for the MFRC522 module. Since the MFRC522 chip can be connected through multiple interfaces (SPI, USART, I2C) this driver itself must be modular. The lower modules handles adressing, reading and writing registers of the MFRC522 over various interfaces. The upper module, which uses abstract read_register and write_register API handles the business logic of the driver.

Abstract API

The biggest design challenge for this driver is the abstract API. This API has to be abstract enough so that drivers for various readers implementing it can be written, however, it should also be easy to comprehend, should adhere to the ISO/IEC 14443-3 standard and, ideally, easy to implement. Naming convention will be similar to the one used previously: 'A'-related functions end with A, B-related functions end with B, common functions end with AB.

Transmission methods in ISO/IEC 14443-3 parts A and B are different enough that they won't share any code. It is therefore possible to design API only for part A and easily add part B later.

Part A of the standard defines transmission of 3 different types of frames:

- Short frame: transmits 7 bits.
- Standard frame: Used for data exchange and can transmit several bytes with parity.
- Bit oriented anticollision frame: 7 byte long frame spit anywhere into two parts. First part is transmitted by the PCD, second part is added by the PICC. It is used during bit-oriented anticollision loop.

ISO/IEC 14443-3 specifies different communication methods (different modulation type / index, different encoding) for part A and B. This driver should support setting these modes, various other communication parameters within these modes as defined by the standard, and should also be able to advertise its capabilities. Communication speeds can't be arbitrary and are defined by ISO/IEC 14443-4 as `1 etu = 128 / (D x fc)`, where etu is the elementary time unit (duration of one bit), fc is the carrier frequency (defined in ISO/IEC 14443-2 to be $13.56MHz \pm 7kHz$) and D is integer divisor, which may be 1, 2, 4 or 8. This paradoxically means that increasing the divisor also increases the communication speed.

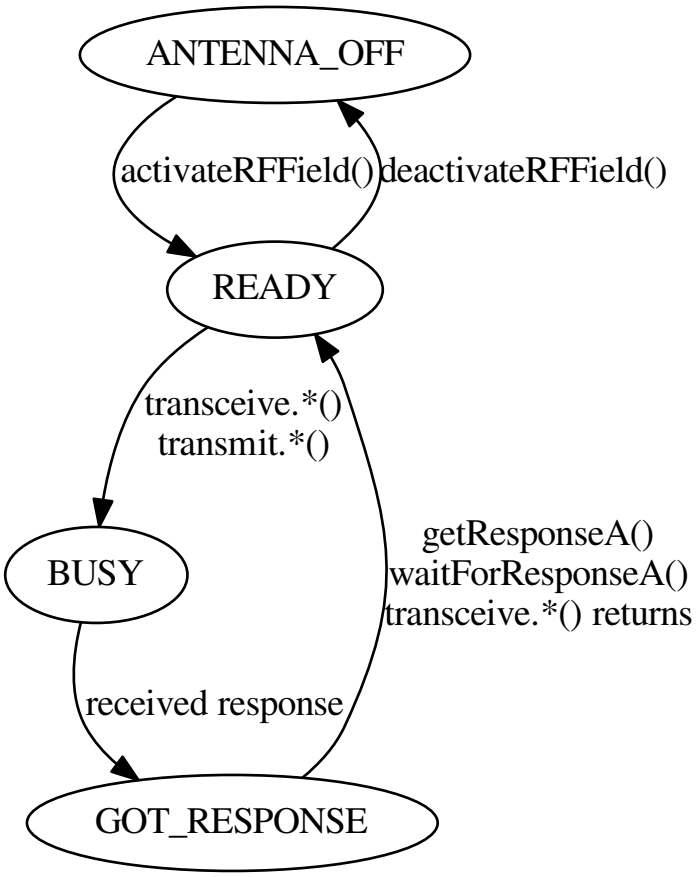
The readers also usually support a number of extended features, not covered by the ISO/IEC 14443 standard. For example, the MFRC522 is able to perform a Mifare authentication using its crypto unit, or a self-test. Upper layers which know how to use these extended features should have access to them, but they should not clutter the main API. That is why each extended feature will have a globally assigned number (in the global abstract header). Then other layers could use these numbers to invoke the extended feature, passing in a "parameter structure". These structures are also defined globally in the abstract header (although in a different file) to allow their re-use.

Ofted PCDs have a maximum data size they can handle at once. ISO/IEC 14443 standard takes this into account and defines "protocol chaining", a method to send large data units in multiple smaller frames. The upper library handles this, but for it to know whether to use chaining the PCD must be able to report maximum frame size it can handle.

Taking the above into consideration the API may look like this:

- `activateRFFieldAB(Pcd*)`: Turns the antenna on and creates an unmodulated RF field
- `deactivateRFFieldAB(Pcd*)`: Turns off the antenna
- `getSupportedParamsAB(Pcd*)`: Returns the supported communication parameters and their combinations
- `getSupportedFrameSizesA(Pcd*)`: Returns the maximum supported frame size, both for transmit and receive operations.
- `setParamsAB(Pcd*,PcdParams*)`: Sets the communication parameters.
- `transceiveShortFrameA(Pcd*,uint8_t data,Frame* response)`: Sends a short frame, waits for a response and returns it
- `transceiveStandardFrameA(Pcd*,uint8_t* buffer,uint8_t length,Frame* response)`: Sends a standard frame, waits for a response and returns it. TODO CRC bullshit.
- `transceiveAnticollFrameA(Pcd*,uint8_t* buffer,uint8_t length,uint8_t valid_bits,Frame* response)`: Sends an anticoll frame, waits for a response and returns it.
- `transmitShortFrameA(Pcd*,uint8_t data)`: Sends a short frame. Returns as soon as possible.
- `transmitStandardFrameA(Pcd*,uint8_t* buffer,uint8_t length)`: Sends a standard frame. Returns as soon as possible.
- `transmitAnticollFrameA(Pcd*,uint8_t buffer,uint8_t length,uint8_t valid_bits,Frame* response)`: Sends an anticollision frame. Returns as soon as possible.
- `waitForResponseA(Pcd*,Frame* response)`: This function blocks until either the response is received or timeout occurs. The timeout is counted from the start of the last transmission as defined in the standard.
- `getResponseA(Pcd*,Frame* response)`: This function will return the last received response. It may be used only once per received response.

Abstract API State transitions



Driver-specific API

So far only abstract API encapsulated in an `Pcd` structure was described. This driver-specific API will be used by the board-specific initialization code to initialize the `MFRC522` itself and construct the `Pcd` structure. Implementation of this API is straightforward and will not be covered here.

Threading and memory management

This library will be thread safe. It will carry sole responsibility for allocating and destroying `Pcd` structures. Basically the same rules apply as for `ISO/IEC 14443A` protocol library.

Contents:

1.7 Server/controller communication protocol

1.7.1 Design requirements

The server/controller communication protocol must facilitate reliability, security, extensibility and ease of development and deployment, as defined in Requirements. This led to the following decisions:

Statelessness and idempotence

In order to keep the protocol simple, yet reliable, communication should be stateless and all messages should be idempotent. This allows for retrying anything that failed for any reason, at any time.

Simplicity

The protocol should be simple. This includes not only low complexity of the protocol's states and messages, but also simplicity in message parsing on any device and in any programming language. Specifically, parsing must be efficient on embedded devices with low CPU frequency and memory.

Extensibility

The protocol must be easily extensible. Additionally, a mechanism for seamlessly transitioning to a newer version in a live system should be provided. These must not interfere with the simplicity requirement.

Security

We do not want to rely on security mechanisms provided by lower layers (as they may or may not be adequate, or present). Therefore the application layer of the protocol must provide sufficient authentication and secrecy.

Built on standard, well-known technologies

Code is a liability. Our code has our bugs, requires specific knowledge, and is our problem. Therefore the protocol should reuse existing code and technologies wherever appropriate.

Note that fulfilling these requirements is not at all trivial, as some, especially extensibility vs. simplicity, or security vs. simplicity, may end up contradicting each other.

Statelessness and idempotence are not strictly required, as reliability may be achieved in different ways, too. However, as shown in the following sections, statelessness is a very useful approach, as it allows to keep things simple even in the face of requirements that would otherwise need significant complexity.

1.7.2 Protocol design

Alternatives considered

The original suggestion was a connection-oriented protocol, where either the server or the controller could initiate communication. This would abstract away details like maximum packet size, handle lost packet retransmissions, and allow for pushing rule or firmware updates from the server side. However, this approach presents multiple problems, such as

- high server failover cost: in a stateful protocol, connection state would either have to be synchronized among multiple servers, which is impractical, or all communication would have to be restarted from the beginning on failure;
- added reliability only at the cost of added complexity: again because of problems with server state;
- more complexity on the server: the server would need to keep track of active controllers, and who has said and heard what;
- extra CPU cycles, flash and memory usage on embedded devices: while an efficient implementation of these abstractions might be unnecessary or could be written, nonexistent code is easier to write, more efficient, and has fewer bugs.

Chosen design: overview, rationale

The chosen communication protocol is connectionless and stateless, and all information exchange is of the form "controller request → server response". All requests (including retries) can be served independently of any other past, present or future requests from this or another controller.

This implies that there is in fact no strict requirement to serve all requests, or any particular subset of requests, by a single server. Multiple server instances can be used as long as the data they need can be synchronized (and even for this synchronization, eventual consistency is sufficient). Therefore, the system may be configured to use several servers, and controllers are expected to send requests, including retries, in a more or less round-robin fashion. ("More or less" means that the controller is allowed to cache "server dead" information in order to skip dysfunctional servers.)

The round-robin scheduling is particularly useful for retries: if there is a problem with a specific server or a specific part of the network, the controller will simply re-send the request to a different server until a good response is received. Therefore the probability that no server can be reached can be significantly reduced by deploying multiple servers in different parts of the network.

Under normal circumstances the server/controller communication is not latency-sensitive, so the round-robin retries approach does not pose a latency problem. Therefore the controllers use round-robin with generous timeouts and exponential back-off to avoid network congestion.

A "bad" response (such as one that cannot be parsed, or an error) is treated the same as if no response were received (except for possibly different timeouts, logging and such), i.e. a retry is sent to the next server. This allows for uniformly handling all kinds of transient and permanent problems with the server, network or other resources.

Live system upgrades

A welcome consequence of the message independence and round-robin retries for all errors is that even if a server cannot parse a controller's request, or a controller cannot parse a server's response, the controller will simply retry with a different server. Therefore in order to transition to an incompatible protocol version, all that is needed is deploying servers with both the "old" and the "new" protocol, and the controllers will simply retry until they find a compatible server. Together with the fact that the server can automatically deliver firmware updates, and that controllers report their firmware version to the server, this makes any and all online system upgrades trivial and fully automatic.

1.7.3 Network stack

The standard network stack is used: Ethernet (IEEE 802.3) as the physical and data link layers, IP as the network layer and UDP as the transport layer. For IP, both IPv4 and IPv6 are supported, and standard ARP or NDP, respectively, is supported for network to link address resolution. IP addresses may be configured statically or obtained via DHCP.

UDP vs. TCP

A standard TCP implementation is available for all devices we will use (it is even bundled with the real-time OS used for the embedded devices). Therefore it seems like using TCP would provide benefits at no additional cost. However, as our protocol is stateless and packet-oriented, and manages retransmissions on the application layer, the only benefit of TCP would in fact be unlimited "packet" length (as opposed to 64kB for UDP [@UDP]), and other than that we would end up emulating a UDP-like service on top of TCP if we chose to use it.

While the unlimited message size looks useful, it is in fact not that helpful – the only messages that do not fit into a single UDP packet are rule database and firmware blobs, and for these it is more efficient to deliver them in explicit chunks, so that the transfer of these large files does not need to start over in case something goes wrong.

Therefore, as the benefits of TCP are in our case not worth the TCP overhead and flash space on embedded devices is limited, we have chosen to use UDP.

1.7.4 Message types, controller behavior {#protocol:messages}

Controllers are expected to download a local copy of the rules database and query that instead of contacting the server whenever access is requested. They send access logs, report their status, and request updates of the rules database and firmware.

As all communication must be initiated by the controller, it must periodically contact the server in order to find out if an updated rules database or firmware is available.

All responses have a response status tag, the value of which is one of `OK`, `ERROR` (permanent error), `TRY_AGAIN` (transient error). Any non-`OK` response must be treated as if the response did not arrive (i.e. usually a retry as in section \ref{protocol:overview} is necessary), except for possibly different timeouts, logging or scheduling. In the following, only `OK` responses are shown.

Note: The following describes the "semantic" data types. The details of the encoding are specified below. The type "byte string" represents a binary-safe string, or an array of bytes of arbitrary length.

Currently, the following message types are recognized:

`PING`: keepalive, DB and FW version info

Contacts the server to report current status and request info about updates. Also used to adjust controller time.

`PING` request:

Field Type Description

---

time integer what time the controller thinks it is db_version integer version of the rules database currently in use fw_version integer currently running firmware version

`PING OK` response:

Field Type Description

---

time integer server time db_version integer newest available version of the rules database fw_version integer newest available firmware version

The controller is expected to adjust its clock to match the server time.

`ALOG`: transfer access logs

Sends access logs to the server.

Controllers attempt to send access logs as soon as possible, but in order not to lose them, they are saved to the SD card until the server confirms they have been written to disk.[^capacity] Logs may be sent in multiple batches if needed.

[^capacity]: We recommend at least 4GB SD cards in order to have enough space for flash wear leveling. As each log record is about 20-30 bytes (depending on the encoding), at a rate of 1 access per second (which is somewhat overstated) it would take about 5 years to run out of space.

`ALOG` request:

Field Type

---

records array of `log_record`s (defined below)

Table: `log_record` structure

Field Type Description

---

time integer timestamp card_id byte string card that requested access allowed boolean was access granted?

`ALOG` `OK` response: All sent records have been written to disk. (Response body empty.)

`XFER`: transfer a file chunk {#protocol:xfer}

Firmware and rule database updates are treated as opaque binary blobs by the `XFER` command. They are identified by type and version. In order to trivially support incremental downloading and arbitrary chunk sizes, the controller explicitly requests the offset and length of the chunk. The same version must always refer to an exactly identical blob (if it exists), even if requested from a completely independent server.[^identical] The server may return a smaller chunk, but never longer. A chunk of length 0 indicates end of file.

[^identical]: See section \ref{impl:fun:fileversion} for notes on how we implemented this. From the protocol's viewpoint, versions must be treated as opaque integers with this and only this guarantee.

`XFER` request:

Field Type Description

---

filetype enum `DB` and `FW` currently supported fileversion integer same version $\implies$ same contents offset integer offset from the beginning of the blob length integer

`XFER` `OK` response:

Field Type Description

---

length integer may be less than requested chunk byte string the file chunk contents

The server will return a `TRY_AGAIN` error if the file was not found. This would usually happen because one server already received and processed an update and another one is behind. The controller will simply retry until it finds a ready server.

Note: These are the only responses longer than a few bytes. The server will send whatever size it is asked for (up to the generous packet size limit). It is each controller's responsibility not to ask for chunks that may result in replies that are too long for it to process. This is to allow maximum efficiency with controllers with different capabilities.

`CRITICAL`: report a critical problem {#protocol:critical}

Used to report a critical problem, upon which the server should take immediate action.

`CRITICAL` request:

Field Type Description

---

code enum error code message optional text string details of the error, if any

Currently the only recognized codes are `LOCK_FORCED_OPEN` (a physical lock was opened without permission) and `READER_NOT_RESPONDING` (a reader is not responding correctly even after multiple restarts), but we assume that more uses will emerge when preparing for real-world deployments.

`CRITICAL OK` response: Acknowledged, action taken. (Response body empty.)

`ASK`: ask if access should be granted now

Because of the potentially high latency of roundtrips, local evaluation rather than querying the server should be used in production. However, we include this for special cases and as a fallback.

`ASK` request:

Field Type Description

---

card_id byte string card that requested access

Whether access should be granted is a function of identity, time and PoA (for details see chapter \ref{rules}). In this case, this card's identity, the current (server) time and the PoA associated with this controller are used.

`ASK OK` response:

Field Type Description

---

allowed boolean do we allow access?

`ECHOTEST`: echo for testing purposes

Echoes the request body. This is helpful in integration testing. Live deployments are recommended to run a process that will act as a controller sending `ECHOTEST` (and possibly other) requests and report any problems. (Such a process is run by default – see section \ref{deadaux}.)

1.7.5 Packet format

Record encoding {#protocol:cbor}

All requests and responses, as well as the outer packet envelope, are "records", i.e. small key-value mappings with fixed key names and types. Therefore we originally wanted to simply transmit "C structs" (i.e. binary blobs with fixed offsets for fields) and hard-code field offsets in the server and controller firmware. However, this approach has multiple disadvantages:

- Any extension would be an incompatible change, and therefore would require the full upgrade procedure as described in section \ref{protocol:live-upgrades}. While this procedure is simple, when it is running, the system requires more servers to achieve the same level of redundancy; and it may make administrators nervous.
- We may parse a packet incorrectly without noticing, if the length matches.
- When the length does not match, we don't know anything more specific than "parsing failed".
- The blob is not self-describing, and therefore nothing is known about it without the context of the outer envelope specifying the version and the description of fields for this version.

Especially the concerns around parsing errors are significant enough to justify a self-describing encoding. Therefore we need an encoding with the following properties:

- self-describing: key names and types must be present in the the encoded data
- expressive: it must be possible to include all the necessary types and arbitrarily nest them as arrays or sub-records; optional fields must be supported
- binary-safe: able to transmit arbitrary binary data (e.g. card IDs or file chunks) without the need for extra encoding
- not incompatible by default: when a backwards-compatible change is introduced (such as adding a new optional field, or removing a field that was optional), old and new code must be able to communicate without change
- suitable for embedded devices: encoding and decoding must be fast, using small code size and producing small messages
- standard, with existing libraries: our code is our problem – the less code we write, the less code we will need to maintain in the future

These requirements are perfectly fulfilled by the Concise Binary Object Representation (CBOR, see [@CBOR]) – a data format designed for communicating with constrained nodes. We use arrays of CBOR semantically tagged items to represent records.[^duplicate] (These are equivalent to arrays of $(\textit{tag}, \textit{data})$ pairs, where $\textit{data}$ is strongly typed.) Unknown tags are ignored and from the parsing viewpoint all fields are optional. In this way the only thing that a server and a controller must have in common to communicate are the tag interpretations (which makes sense, if they want to use the values for something useful).

[^duplicate]: If a duplicate tag is encountered, it is considered an error. In addition to serving as a sanity check, this might prevent some overflow-related attacks.

Requests, responses {#protocol:requests-responses}

For all requests and responses, the record as specified in \ref{protocol:messages} is tagged by a semantic tag for the corresponding message type, and in case of response records this is in turn tagged by the response status.

"Envelope" – version, addressing, encryption

The outer layer of the messages (common to requests and responses) provides addressing and encryption. It is a record with fields as specified in table \ref{table:protocol:envelope}. The encoded record is prepended with a 4-byte "magic number" containing the bytes $[68, 69, 65, 68]$ ('DEAD' in ASCII) identifying this as a Deadlock message.

Field Type Description

Version identifier integer unknown version must be ignored Controller ID integer addressing Nonce 24 bytes random bytes Payload byte string encrypted request/response

Table: Message record. \label{table:protocol:envelope}

Version identifier : Packet must be considered invalid if this does not match a known version. This is to support live system upgrades, as detailed in section \ref{protocol:live-upgrades}.

Controller ID : Unique identifier of the sender or intended recipient. Serves as addressing. Including a form of addressing on the application layer decouples "logical" addressing from "physical" (i.e. network) addressing, thereby allowing Deadlock to function over NAT, with broadcast/multicast/anycast IP addresses, and such.

Nonce : Randomly generated bytes. Matches a response to a request: when a request nonce is x, the associated response's nonce must be $x \oplus 1$. Used as detailed in section \ref{protocol:security}.

Payload : Request/response, encoded according to section \ref{protocol:cbor}. Encrypted with the key for the given controller using the nonce, as detailed in section \ref{protocol:security}.

Note: Maximum message size (when encoded and encrypted) is 63kB (in order to comfortably fit into a UDP packet).

1.7.6 Security {#protocol:security}

Security is complicated. While libraries implementing cryptographic primitives exist, they usually do not make securing an application particularly easy: the developer must be aware of what needs what kind of security; which primitives (such as cipher, cipher mode, checksums, signatures) are suitable for which use case, what they promise, what their weaknesses are and whether they are a problem in the given use case; she must consider potential side channel attacks, replay attacks, and such; and she must ensure other developers are aware of all these considerations. As the numerous vulnerability reports published each month signify, this is no easy task.

Short of locking one's computer in a closet without electricity, the best way to secure a system is to leave it to an expert. Luckily, in 2013 the NaCl library interface specification [@NaClDoc] and several implementations were published, with the aim of providing developers with a simple, "sane defaults" crypto toolkit. See [@NaClSecurity] for a discussion of the impact of such a library.

In Deadlock, we assume operation over untrusted networks, and we must resist both passive and active attacks. Therefore we encrypt and authenticate all messages from/to a given controller with a device-specific symmetric key, using NaCl's `secret_box(nonce,key,payload)` function, which promises secrecy and integrity provided the nonce is not used more than once [@NaClDoc]. We construct the nonce by generating 24 random bytes,[^random] which ensures negligible collision probability (quick birthday paradox approximation says the probability reaches 50% after more than 10^{28} packets, which is a lot). Symmetric cryptography was chosen for performance, but once the actual controller hardware and firmware exists, we are planning to run benchmarks and switch to asymmetric cryptography if possible, in order to avoid the need to copy the secret to more than one place.

The default NaCl primitives in NaCl are the Salsa20 stream cipher for symmetric encryption and the Poly1305 MAC for message authentication. As detailed in [@NaClCrypto], these are secure and performant without depending on any form of hardware acceleration, which goes well with our requirements.

[^random]: "Random" in this case does not mean cryptographically secure randomness – nonces may be predictable (they are sent in cleartext along with the payload anyway), the only requirement is a uniform distribution to ensure low collision probability. The fact that NaCl does not require a source of good randomness is in embedded environments very welcome.

Security guarantees

Provided a nonce is not used more than once, `secret_box(nonce,key,payload)` guarantees

- **secrecy**: it is infeasible to decrypt a message without knowledge of the key;
- **integrity**: if a message is decrypted successfully, no accidental or purposeful third party modification of the nonce or the encrypted payload can have occurred;
- **resistance to timing attacks**: the implementations try to always perform the same amount of work.

Furthermore, the protocol's idempotence and use of nonces **prevents replay attacks**: if an attacker attempts to replay a request to a server, nothing bad will happen as all requests are idempotent; if she replays a response to a controller, its nonce will not match any of the responses the controller is currently expecting and therefore it will ignore the fake response.

1.8 The Deadlock server

Contents:

1.8.1 `deadapi` – the HTTP API for the outside world

TODO

1.9 Controller

This is a (usually embedded) device with network connectivity to the server. Its job is to open and close single door it controls when it receives information about a card from the Reader. It also tells Server about everything that is happening with the door (like usage of a manual lock override (opening doors with a physical key) or manual door override (like kicking the door out)). It can be connected to up to 2 readers (possibly more with a proper extension board).

This device will be gradually developed, with 2 models. First model (Controller Model A) will be able to function only if it has an active connection to the server, later more advances model (Controller Model B) will be able to perform it's decisions offline if the server is down.

It will also be extensible with 'Controller Extension Boards', that can provide extended functionality.

1.10 Reader

The Reader is the small box people touch with their cards. It provides simplistic visual and auditory interface (it blinks and beeps). It also reads the card and provides the controller with necessary information for deciding whether to open the door.

The Reader is connected to the Controller using a cable, utilizing a simple serial interface. It will also be developed in more models, Reader will only have the serial interface, Reader+ will also have USB capability, for use with other systems.

1.11 Deadlock front-ends: Web management console, CLI, data importers

All of these tools interact with the server via [its REST API](../server/deadapi).

Contents:

1.11.1 `webui` – the Deadlock web interface

Contents:

Introduction

`webui` is a single-page web application that connects to `deadapi` to provide the following functions:

- access rules management
- point of access management
- system status monitoring
- access logs monitoring

It is written in React.